L Number	Hits	Search Text	DB	Time stamp
1	41	inversion near5 safe	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2003/09/30 20:44
2	58	inheritance near5 priorit$6	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2003/09/30 20:44
3	4	(inheritance near5 priorit$6) and (inversion near5 safe)	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2003/09/30 20:58
4	135	pip same variable$1	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2003/09/30 20:59
5	0	(inheritance near5 priorit$6) and (pip same variable$1)	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2003/09/30 20:59
6	6	(inheritance near5 priorit$6) same variable$1	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2003/09/30 21:02
7	3	inversion$1 and ((inheritance near5 priorit$6) same variable$1)	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2003/09/30 21:55
8	8	(mutual near5 exclusion$1) same (inheritance near5 priorit$6)	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2003/09/30 21:55

US-PAT-NO: 6560627

DOCUMENT-IDENTIFIER: US 6560627 B1

TITLE: **Mutual exclusion** at the record level with **priority inheritance** for embedded systems using one semaphore

DATE-ISSUED: May 6, 2003

INVENTOR-INFORMATION:

NAME COUNTRY	CITY	STATE	ZIP CODE	
McDonald; Michael F.	San Jose	CA	N/A	N/A
Arora; Sumeet	Milpitas	CA	N/A	N/A
Chu; Mark	Cupertino	CA	N/A	N/A

US-CL-CURRENT: 709/103, 709/100 , 709/104

ABSTRACT:

A method for providing mutual exclusion at a single data element level for use in embedded systems. Entries for tasks that are currently holding a resource are stored in a hold list. Entries for tasks that are not currently executing and are waiting to be freed are stored in a wait list. A single mutual exclusion semaphore flags any request to access a resource.

24 Claims, 7 Drawing figures

Exemplary Claim Number: 1

Number of Drawing Sheets: 7

---------- KWIC ---------

TITLE - TI (1):

Mutual exclusion at the record level with **priority inheritance** for embedded systems using one semaphore

Brief Summary Text - BSTX (7):

Furthermore, in most applications, the **mutual exclusion** mechanism must support **priority inheritance**. If a low priority task holds a resource, and a higher priority task requests that resource, the priority of the low priority task should be elevated to that of the high priority task until it task releases the resource. Once the resource is released, priorities should revert to their original levels. In general, it is also desirable for the **mutual exclusion** mechanism to be able to detect and/or prevent deadlock. In a multi-tasking environment several tasks may compete for a finite number of resources. A task requests resources; if the resources are not available at that time the tasks enters the wait state. It may happen that waiting tasks will never again change state, because the resources they have requested are held by other waiting tasks. This situation is called deadlock. For example, deadlock occurs when a first task requests a record held by a second task while the second task is simultaneously requesting a record held by the first task. The result is neither task has its request answered. Such an occurrence could cause the application program or system software to crash.

Detailed Description Text - DETX (2):

A system is described that provides **mutual exclusion** of multiple tasks at the record level with **priority inheritance** and using one semaphore.

Detailed Description Text - DETX (38):

In the foregoing, a system has been described for providing **mutual exclusion** of multiple tasks at the record level with **priority inheritance** and using one semaphore. Although the present invention has been described with reference to specific exemplary embodiments, it will be evident that various modifications and changes may be made to these embodiments without departing from the broader spirit and scope of the invention as set forth in the claims. Accordingly, the specification and drawings are to be regarded in an illustrative rather than a restrictive sense.

US-PAT-NO: 5872909

DOCUMENT-IDENTIFIER: US 5872909 A

TITLE: Logic analyzer for software

DATE-ISSUED: February 16, 1999

INVENTOR-INFORMATION:

NAME	CITY	STATE	ZIP CODE	COUNTRY
Wilner; David N.	Oakland	CA	N/A	N/A
Smith; Colin	Alameda	CA	N/A	N/A
Cohen; Robert D.	Oakland	CA	N/A	N/A
Burd; Dana	Oakland	CA	N/A	N/A
Fogelin; John C.	Berkeley	CA	N/A	N/A
Fox; Mark A.	San Francisco	CA	N/A	N/A
Long; Kent D.	Richmond	CA	N/A	N/A
Burns; Stella M.	San Francisco	CA	N/A	N/A

US-CL-CURRENT: 714/38, 714/47

ABSTRACT:

The present invention logs events which occur in the target software, and stores these in a buffer for periodic uploading to a host computer. Such events include the context switching of particular software tasks, and task status at such context switch times, along with events triggering such a context switch, or other events. The host computer reconstructs the real-time status of the target software from the limited event data uploaded to it. The status information is then displayed in a user-friendly manner. This provides the ability to perform a logic analyzer function on real-time software. A display having multiple rows, with one for each task or interrupt level, is provided. Along a time line, an indicator shows the status of each program, with icons indicating events and any change in status.

30 Claims, 18 Drawing figures

Exemplary Claim Number: 1

Number of Drawing Sheets: 14

---------- KWIC ---------

Detailed Description Text - DETX (26):

Each task has a "priority" which indicates that task's eligibility to control the CPU relative to the other tasks in the system. A task is in the inherited state when its priority has been increased because it owns a **mutual exclusion** semaphore that has **priority inheritance enabled and a higher-priority** task is waiting for that semaphore. **Priority inheritance is a solution to the priority** inversion problem: a higher-priority task being forced to wait an

indefinite period of time for the completion of a lower-priority task. For example, assume that task 30 needs to take the **mutual exclusion** semaphore for a region, but taskLow currently owns the semaphore. Although taskHi preempts taskLow, taskHi pends immediately because it cannot take the semaphore. Under some circumstances, taskLow can then run, complete its critical region and release the semaphore, making taskHi ready to run. If, however, a third task, taskMed, preempts taskLow such that taskLow is unable to release the semaphore, then taskHi will never get to run. With the **priority inheritance** option enabled, a task, such as taskLow, that owns a resource executes at the priority of the highest-priority task pended on that resource, the priority of taskHi in the example. When the task gives up the resource, it returns to its normal priority.

TDB-ACC-NO: NN9507111

DISCLOSURE TITLE: Microkernel Semaphores

PUBLICATION-DATA: IBM Technical Disclosure Bulletin, July 1995, US

VOLUME NUMBER: 38

ISSUE NUMBER: 7

PAGE NUMBER: 111 - 118

PUBLICATION-DATE: July 1, 1995 (19950701)

CROSS REFERENCE: 0018-8689-38-7-111

DISCLOSURE TEXT:

Described is the architecture, analysis, design and implementation of fast synchronization primitives in the IBM* Microkernel Product. Without these mechanisms, only kernel mechanisms are provided to enable applications and/or threads to rendezvous or to protect resources. In the case where resources are often available, the use of kernel mechanisms is often too expensive.
- The function of the synchronization services package will now be described. In addition to any synchronization services provided by a threading package, such as C threads, the microkernel product provides three types of user level synchronization---counting semaphores, exclusion semaphores, and conditions---each of which can be used for synchronization within a single task. Counting semaphores can be used for synchronization between cooperating tasks. The counting semaphores and conditions mechanisms provide timeout capabilities for functions that block. Additionally, the counting functionality semaphores allow for unlimited readers, while providing some basic recovery features. The design ensures that uncontested acquisition of a synchronization structure is as fast as possible since there is no interaction with the microkernel.

The synchronization services use the following data structures:

- o sema_id_t is a pointer to a counting semaphore that permits limited restricted acquisitions and unlimited shared acquisitions.
- o sema_attr_t is a pointer to the semaphore attribute structure. The countfield specifies how many restricted acquisitions are permitted. The type field specifies if the semaphore is to be shared by multiple tasks, and if it is to be recoverable.
- o mutex_id_t is a pointer to a **mutual exclusion** semaphore that permits only one restricted acquisition and no shared acquisitions.
- o mutex_attr_t is a pointer to the **mutual exclusion** semaphore attribute structure.

The policy field specifies which lock management policy will be applied to the mutex. Possible values are MUTEX_BP or MUTEX_BPI. MUTEX_BP sets the policy to basic priority protocol. MUTEX_BPI sets the policy to basic priority protocol with **priority inheritance**. **Priority inheritance** is supported, but may affect performance.

- o cond_id_t is a pointer to a condition variable. cond_attr_t is a pointer to the condition variable attribute structure. The

timeout field specifies the default timeout period for this condition variable. The default for timeout is TIMEOUT_NONE.

If the timeout variable is set to 0, the call will return immediately after requesting a context switch. The mutex field is filled in by the user to specify which mutex is associated with the condition.

In order to provide fast synchronization services, it is important to avoid unnecessary microkernel calls. Therefore, the microkernel semaphore library uses shared memory (either in a single task or between tasks) to house its semaphore, mutex, and condition variables. These are all protected by simple locks (spin locks). When an attempt is made to acquire one of these resources, the spin lock is taken and the data structure in shared memory is analyzed.

If available, the data structure is marked appropriately, the simple lock released and the thread proceeds. If the resource is not available, a kernel call is made to atomically block on a virtual-memory-based condition and release the simple lock. When a mutex or semaphore is released, or when a condition is posted, the simple lock is also taken. The shared memory data structures are again analyzed to determine if there are any threads waiting on this event. If there are, a call is made into the microkernel to post to the appropriate virtual memory condition, unblocking either one or all waiting threads. Finally, the simple lock is released.

The semaphore services provide the following interfaces to the user:

semas_condition_broadcast

Function--Indicates a change in a condition variable to all waiting threads.

Synopsis--kern_return_t semas_condition_broadcast(
  cond_id_t cond);

Description--The semas_condition_broadcast function indicates a status change in the condition cond. The associated mutex must be locked across this call. All threads that are waiting on the condition are awakened.

semas_condition_clear

Function--Terminates use of a condition variable.

Synopsis--kern_return_t semas_condition_clear(
  cond_id_t cond);

Description--The semas_condition_clear function finalizes the use of the condition variable identified as cond. If there are any waiters on the condition, an error will be returned and the condition will not be cleared.

semas_condition_get_attr

Function--Obtains current attributes of a condition variable.

Synopsis--kern_return_t semas_condition_get_attr(
  cond_id_t cond,
  cond_attr_t cond_attr);

Description--The semas_condition_get_attr function fills in the structure referenced by cond_attr with the attributes associated with the condition cond. If cond does not exist, an error is returned.

semas_condition_init

Function--Initializes a condition variable.

Synopsis--kern_return_t semas_condition_init(

```
  cond_id_t         cond,
  cond_attr_t       cond_attr);
```

Description--The semas_condition_init function initializes the condition variable cond with the attributes specified in the attribute structure referred to by cond_attr.

When the condition variable is initialized, the attribute structure can be reused or freed.

semas_condition_set_attr

Function--Sets attributes of a condition variable.

Synopsis--kern_return_t semas_condition_set_attr(

```
  cond_id_t                cond,
  cond_attr_t                   cond_attr);
```

Description--The semas_condition_set_attr function uses the structure referenced by cond_attr, set the attributes (such as default time-out) associated with the already initialized condition cond. Changing the attributes does not affect any threads currently waiting on the condition. The mutex field is not changed.

If cond does not exist, an error is returned.

semas_condition_signal

Function--Indicates a change in a condition variable to a single waiting thread.

Synopsis--kern_return_t semas_condition_signal(

```
  cond_id_t         cond);
```

Description--The semas_condition_signal function indicates a status change in the condition cond. The associated mutex must be locked across this call. The highest priority thread that is waiting on the condition is awakened.

semas_condition_wait

Function--Waits for notification on a condition variable.

Synopsis--kern_return_t semas_condition_wait(

```
  cond_id_t         cond);
```

Description--The semas_condition_wait function waits for notification to occur on condition cond. The default time out period is used for this condition. The associated mutex must be locked by the calling thread and then unlocked and relocked across the wait, even if the default time-out period is zero. Upon successful return, the condition must be retested. Attempts to wait on a freed condition return an error.

semas_condition_wait_timed

Function--Waits with specified timeout for a notification on a condition variable

Synopsis--kern_return_t semas_condition_wait_timed(

```
  cond_id_t         cond,
  timeout_t         timeout);
```

Description--The semas_condition_wait_timed function waits for notification to occur on condition cond. The time out period is specified by timeout. The associated mutex must be locked by the calling thread and then unlocked and relocked across the wait, even if the time-out period is zero. Upon successful return, the condition must be retested. Attempts to wait on a freed condition return an error.

semas_condition_waiters

Function--Returns an indication of whether any threads are waiting on a condition variable.

Synopsis--kern_return_t semas_condition_waiters(

```
  cond_id_t         cond,
```

boolean_t *waiters);
Description--The semas_condition_waiters function returns, in waiters, a boolean_t value indicating whether there are any threads waiting on condition cond. This call is informational only; the state may change upon return.
semas_mutex_clear
Function--Terminates use of a mutual exclusion semaphore.

Synopsis--kern_return_t semas_mutex_clear(
mutex_id_t mutex);
Description--The semas_mutex_clear function finalizes the use of the mutual exclusion semaphore identified as mutex. If the mutex is currently locked, an error is returned and the mutex is not finalized. It is the responsibility of the user to finalize any conditions that may be associated with mutex before finalizing the mutex.
semas_mutex_init
Function--Initializes a mutual exclusion semaphore with specified attributes.

Synopsis--kern_return_t semas_mutex_init(
mutex_id_t mutex,
mutex_attr_t mutex_attr);
Description--The semas_mutex_init function initializes the mutual exclusion semaphore referred to by mutex with the attributes specified in the attribute structure referred to by mutex_attr. Once the mutual exclusion semaphore is initialized, the attribute structure can be reused or freed.
semas_mutex_lock
Function--Acquires a mutual exclusion semaphore.
Synopsis--kern_return_t semas_mutex_lock(
mutex_id_t mutex);

Description--The semas_mutex_lock function acquires a mutual exclusion semaphore identified as the mutex. This request succeeds immediately if no other thread has acquired the mutex. If the mutex can not be acquired immediately, the thread waits. Attempts to acquire a cleared mutex return KERN_INVALID_ARGUMENT.
semas_mutex_unlock
Function--Unlocks a mutex
Synopsis--kern_return_t semas_ mutex_unlock (
mutex_id_t mutex);
Description--The semas_mutex_unlock function releases the mutex.

semas_mutex_waiters
Function--Indicates whether any threads are waiting on a semaphore.
Synopsis--kern_return_t semas_mutex_waiters(
mutex_id_t mutex,
boolean_t *waiters);
Description--The semas_mutex_waiters function returns, in waiters, a boolean value indicating whether there are any threads waiting on mutex. Threads waiting on an associated condition are not considered to be waiting on the mutex. This call is informational only. There is no guarantee that the state remains unchanged upon return.

semas_sema_attach
Function--Attaches a shared semaphore to the current task.

```
Synopsis--kern_return_t semas_sema_attach(
  sema_id_t          sema)
```

Description--The semas_sema_attach function attaches shared semaphore sema to the calling task. A task passes the id (a pointer) of a shared semaphore to another task. The second task calls this routine to attach to the semaphore. When done, the task, which must have read/write access to the memory area where the semaphore resides, detaches itself from the semaphore with a call to semas_sema_detach().

The task that originally initializes a shared semaphore is automatically attached to it.

semas_sema_clear

Function--Terminates use of semaphore.

Synopsis

```
  kern_return_t semas_sema_clear(
  sema_id_t          sema);
```

Description--The semas_sema_clear function finalizes the use of the semaphore identified as sema. If the semaphore is currently acquired by another thread, an error is returned. If the semaphore is a shared semaphore, and there are still tasks attached to it, an error is returned.

Any subsequent lock requests that occur after the semaphore has been de-allocated are returned KERN_INVALID_ARGUMENT.

semas_sema_detach

Function--Detaches a semaphore from the current task.

```
Synopsis--kern_return_t semas_sema_detach(
  sema_id_t          sema);
```

Description--The semas_sema_detach function detaches the shared semaphore sema from the current task.

semas_sema_init

Function--Initializes a semaphore with specified attributes.

```
   Synopsis--kern_return_t semas_sema_init(
  sema_id_t          sema,
  sema_attr_t        sema_attr);
```

Description--The semas_sema_init function initializes the semaphore referred to by sema with the attributes specified in the structure referred to by sema_attr. If the semaphore is shared, the calling task automatically attaches to the semaphore. Once the semaphore is initialized, the attribute structure can be reused or freed.

semas_sema_lock

Function--Acquires a semaphore.

```
   Synopsis--kern_return_t semas_sema_lock(
  sema_id_t                 sema,
  timeout_t                 timeout,
  int       lock_mode);
```

Description--The semas_sema_lock function acquires a semaphore if lock_mode has the value LOCK_RESTRICTED. This function acquires the semaphore in restricted mode. This request succeeds immediately if no other thread has requested or acquired the semaphore in read-only mode or exclusive mode, and the number of restricted acquisitions does not exceed the limit specified when the semaphore was allocated. If the semaphore can not be acquired within the specified time period, an error is returned.

If lock_mode has the value LOCK_READONLY. This function acquires the semaphore in read-only mode. This request succeeds immediately if no other thread has requested or acquired the

semaphore in restricted mode or exclusive mode. If the semaphore cannot be acquired within the specified time period, an error is returned. If lock_mode has the value LOCK_EXCLUSIVE, this function acquires the semaphore exclusively. This request succeeds immediately if no other thread has acquired the semaphore. If the semaphore cannot be acquired within the specified time period, an error is returned. If the semaphore is a recoverable semaphore, the id of the locking thread is added to the holders list for the semaphore.

semas_sema_recovered

Function--Indicates that damaged data related to a semaphore has been repaired.

Synopsis--kern_return_t semas_sema_recovered(

```
sema_id_t          sema);
```

Description--The semas_sema_recovered function indicates that previously damaged semaphore-protected data has been restored to a known state. The semaphore remains locked after a call to semas_sema_recovered. The recommended sequence is to acquire the damaged semaphore exclusively, repair the data, indicate the data is no longer suspect (this call), and unlock the semaphore. If a thread terminates while holding a recoverable semaphore, an attempt is made to remedy the situation.

(Recovery of a read-only semaphore lock is straightforward: the count is adjusted and any blocked threads are awakened as if a normal semas_sema_unlock occurred.) Recovery of an exclusive semaphore lock proceeds as follows: the semaphore is marked as damaged, the count is adjusted and subsequent acquisitions will succeed, but a status of SEMA_DAMAGED is returned from the lock routine that indicates that semaphore data might be inconsistent. If the user is able to restore the protected data to a known state, the damaged status might be cleared by a call to the semas_sema_recovered routine.

semas_sema_unlock

Function--Releases a lock on a semaphore.

Synopsis--kern_return_t semas_sema_unlock(

```
sema_id_t          sema);
```

Description--The semas_sema_unlock function releases a lock on the semaphore sema. If the semaphore being unlocked is a recoverable semaphore, the id of the unlocking thread is removed from the holders list of the semaphore. If the task and thread id of the unlocking thread are not on the holders list, an error is returned and the thread is not unlocked.

semas_sema_waiters

Function--Indicates if threads are waiting on a semaphore.

Synopsis--kern_return_t semas_sema_waiters(

```
sema_id_t                sema,
boolean_t                *waiters);
```

Description--The semas_sema_waiter function indicates whether any threads are blocked waiting for a semaphore. There is no guarantee that the state remains unchanged upon return.

* Trademark of IBM Corp.

Secrets Act, 18 U.S.C. 1905.

L Number	Hits	Search Text	DB	Time stamp
1	41	inversion near5 safe	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2003/09/30 20:44
2	58	inheritance near5 priorit$6	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2003/09/30 20:44
3	4	(inheritance near5 priorit$6) and (inversion near5 safe)	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2003/09/30 20:58
4	135	pip same variable$1	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2003/09/30 20:59
5	0	(inheritance near5 priorit$6) and (pip same variable$1)	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2003/09/30 20:59
6	6	(inheritance near5 priorit$6) same variable$1	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2003/09/30 21:02
7	3	inversion$1 and ((inheritance near5 priorit$6) same variable$1)	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2003/09/30 21:55
8	8	(mutual near5 exclusion$1) same (inheritance near5 priorit$6)	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2003/09/30 21:55